다중 공유 자원을 위한 프로세스 대수
(Process Algebra for Multiple Shared Resources)

유 희 준 † 이 기 흔 †† 최 진 영 †††
(Hee-Jun Yoo) (Ki-Huen Lee) (Jin-Young Choi)

요 약 본 논문에서는 다중자원(multiple resource)을 사용하는 시스템의 명세와 검증을 위한 프로세스 대수 ACSMR(Algebra of Communicating Shared Multiple Resources)을 정의한다. ACSMR은 프로세스 대수 기반의 정형기법(formal methods)인 ACSR에 다중자원의 개념을 확장한 것이다. 명세와 검증의 예로 실시간 시스템의 스케줄링 기법의 하나인 Earliest-Deadline-First(EDF)를 멀티프로세서하에서의 시스템의 행동 명세와 다중 포트를 가진 레지스터를 이용한 수퍼스칼라 프로세서의 타이밍 특성과 자원 제한을 묘사하기 위한 명세방법을 제시한다.

Abstract In this paper, we define a Process Algebra ACSMR(Algebra of Communicating Shared Multiple Resources) for system specification and verification using multiple resources. ACSMR extends a concept of multiple resources in ACSR that is a branch of formal methods based on process algebra. We'll show that two specification and verification examples. One is the specification of system behavior in multiprocessor using EDF(Earliest-Deadline-First) which is a scheduling algorithm of a real-time system. The other is the specification of describing timing analysis and resources restriction in a super scalar processor using multiple ports registers.

1. 서 론

현재 발달된 하드웨어 기술로 인해서, 여러 작업들이 다중 프로세서 하에서 수행이 되고 있다. 작업을 수행하기 위해 많은 스케줄링 알고리즘들이 사용되고 있으며, 각각의 알고리즘들은 단일 프로세서와 다중 프로세서에서 스케줄링 가능성이 차이를 보이게 된다. 또한 읽기 접근에 대해서 한번에 여러 명령어에서 레지스터를 공유할 수 있는 다중 포트를 가진 레지스터를 이용한 수퍼스칼라 프로세서가 실용화되고 있다. 따라서, 이러한 다중 자원을 사용하는 시스템을 정확히 명세, 검증해야 한다. 여기서는 다중 자원을 사용하는 명세, 검증하기 위해서 ACSMR을 정의하였다.

이 논문에서는 CCS[1](Calculus of Communicating System)의 일종인 ACSR[2](Algebra of Communicating Shared Resource)에 다중자원을 정의한 환경의 개념을 첨가한 ACSMR(Algebra of Communicating Shared Multiple Resources)을 제시했다. ACSR은 CCS에 시간과 자원의 개념이 첨가된 기법으로 시스템을 애매 모호함이 없이 정확하게 명세를 할 수 있다. ACSMR은 여기에 다중자원의 개념을 첨가해서, 다중자원을 사용하는 시스템을 보다 정확하고 효과적으로 명세할 수 있다. ACSMR 명세는 ACSR에서 다중 자원을 사용하는 경우에 과도한 선택 연산을 수행해야하는 단점을 보안하기 위해서 다중 자원에 대해서 각각의 레이블을 주어서 개별적인 자원으로 행동하도록 명세한 것이다. 사용한 프로세서와 추론 규칙은 ACSR과 같다.

여기서는 다중자원을 사용하는 시스템 명세의 예로 두 가지를 제시했다. 하나는 실시간 시스템의 스케줄링 방법중의 하나인 Earliest-Deadline-First 방법을 멀티프로세서(multiprocessor)하에서 명세, 검증한 것이고, 다른 하나는 다중 포트를 가진 레지스터를 이용한 수퍼

· 한국학술진흥재단의 신진교수 공모과제 연구비의 지원을 받았습니다.(997-003-E0031)
† 학생회원 : 고려대학교 컴퓨터학과
hyoo@formal.korea.ac.kr
†† 비 회 원 : 고려대학교 컴퓨터학과
klee@formal.korea.ac.kr
††† 종신회원 : 고려대학교 컴퓨터학과 교수
choi@formal.korea.ac.kr
논문접수 : 1999년 2월 4일
심사완료 : 2000년 1월 14일

스칼라 프로세서의 타이밍 특성과 자원 제한을 묘사하기 위한 명세와 검증을 한 것이다.

EDF의 예에서는 여러 개의 태스크들이 두 개의 CPU에서 병렬로 처리하는 시스템을 명세, 검증하면서, 단일 CPU를 사용했을 경우에는 교착상태(deadlock)가 되지만, 두 개의 CPU를 사용했을 경우에는 스케줄링이 가능한 태스크를 찾아서 이것을 명세, 검증하였다.

수퍼스칼라 프로세서의 예에서는 ACSMR을 이용해서 파이프라인 수퍼스칼라 프로세스에서 일련의 명령어로 된 프로그램을 실행하는데 있어 타이밍 특성과 자원 제한을 분석하기 위한 방법을 제시함으로써 ISA (Instruction Set Architecture)레벨의[3] 기술을 늘리고자 하는 것이다.

이 논문의 구성은 2,3절에서는 ACSMR의 기본적인 문법과 그 동작 의미를 설명하고, 4절에서는 첫 번째 적용 예인 EDF를 멀티프로세서 하에서 시스템 명세를 설명하겠다. 5장은 두 번째 적용 예인 수퍼스칼라 프로세서에서 모델링할 모델 프로세서인 ToyP와 그 명령어 명세를 설명한 후, 마지막으로 6장에서 결론을 맺는다.

2. ACSMR의 Syntax

ACSMR은 CCS에 기반을 한 프로세스 대수로 시간, 자원, 우선 순위, 동시성 등 실시간 시스템에 필요한 여러 개념을 포함한다.

한 개의 ACSMR 프로세스는 $E \triangleright P$ 프로세스의 실행은 레이블이 있는 트랜지션 시스템(labelled transition system)으로 정의된다. 한 예로 프로세스 $E \triangleright P$가 다음과 같은 행태(behavior)을 보일 수 있다. 여기서 E는 사용 가능한 자원이 명시되어 있는 환경을 의미하며, 그 표현은 $\{(n, r)\}$과 같이 나타낸다. 여기서 r은 공유되는 자원을 의미하며, n은 부분 자원의 개수를 의미한다. 또한, P는 ACSMR 프로세스를 의미한다. 다음은 레이블이 있는 트랜지션 시스템의 한 예이다.

$$E \triangleright P_1 \xrightarrow{\alpha_1} E \triangleright P_2 \xrightarrow{\alpha_2} E \triangleright P_3 \xrightarrow{\alpha_3} \cdots$$

즉, 처음에는 $E \triangleright P_1$이 α_1이라는 액션을 실행하고 $E \triangleright P_2$라는 프로세스가 되는데, 이 프로세스 $E \triangleright P_2$는 α_1라는 액션을 실행한다. 이때 프로세스 P_1과 P_2의 사용 가능한 자원이 E에 의해 정의되었을 경우에만 실행 가능하다. 즉, $\{(1,cpu)\} \triangleright \{(1,io)\}.P$는 실행 불가능하다. 두 프로세스 사이의 통신은 이벤트와 역(inverse) 이벤트로 이루어지는데, 이미 확립된 통신에 더 이상의 동기화를 막기 위해 이 두 개의 액션은 내부 액션으로 변환되어 외부로부터의 간섭을 금지시킨다. ACSMR에서는 어떤 액션에 우선 순위를 관련시켜서 우선 순위가 제공되는데, 예를 들어 $a.P + b.Q$에서 a의 우선 순위가 b보다 높으면 a가 먼저 실행된다. 다음의 문법은 ACSR 프로세스를 정의되고, ACSMR의 프로세스는 $E \triangleright P$로 정의된다. 이때, E는 환경, P는 ACSMR 프로세스이다.

$$P ::= \mathrm{NIL} \mid A{:}P \mid (a,n).P \mid P+Q \mid P \| Q \mid [P]_I \mid P \backslash F \mid recX.P \mid X$$

NIL은 정지하여 아무런 액션을 하지 않는 프로세스를 의미한다(즉, 교착상태에 빠진 프로세스를 의미한다). 두 종류의 액션을 표시하는 두 개의 연산자(operator)가 있다. 첫 번째로, $A{:}P$는 자원을 소비하는 액션 A를 주어진 단위시간 내에 실행하고 다음 프로세스인 P로 진행하는 것이다. ACSR에서는 액션 A를 자원 r과 우선순위 p의 쌍 (r,p)로 나타내지만, ACSMR에서는 여기에 사용되는 부분 자원의 개수를 더한 세 개의 순서쌍(triple)으로 나타낸다. 예를 들어, $(cpu,1,1)$은 공유자원인 cpu는 우선순위를 1로 가지고, 한번 수행 시에 한 개의 부분자원을 사용함을 의미한다. 특히, 공유자원이 부분자원 하나만을 사용할 경우에는 부분자원의 개수는 생략해서 $(cpu,1)$로 간략히 표현하였다. 이에 반해 $(a,n).P$는 시간의 진행이 없는 이벤트 (a,n)를 실행하고 P로 진행하는 것이다.(여기서 a는 자원을 의미하고 n은 우선 순위이다.) 이 두 가지의 차이점은 이벤트의 경우 시간이 고려되지 않는다는 점이다. 선택(choice) 연산자 $P+Q$는 비결정성(nondeterminism)을 표현한다. 주변환경의 제약조건에 의해 프로세스 P가 선택되거나 프로세스 Q가 선택된다. 병행(concurrent) 연산자 $P \| Q$는 병렬 프로세스를 표현하게 해 준다. 즉 프로세스 P와 Q가 동시에 진행함을 나타낸다. 폐쇄(close) 연산자 $[P]_I$는 프로세스 P가 집합 I에 정의된 자원을 독점함을 표시한다. 제한(restriction) 연산자 P/F는 P의 행태를 제한한다. 즉 프로세스 P는 집합 F안에 정의된 이벤트는 실행할 수 없음을 표현한다. $recX.P$는 반복(recursion)을 나타내며 같은 동작이 무한히 반복되는 것을 표현한다.

3. ACSMR의 Operational Semantics

이 절에서는 ACSMR의 중요한 한 특성중의 하나인 동작원리(operational semantics)를 알아본다.

ACSMR의 의미는 두 단계로 정의가 되는데, 여기서는 첫 번째 단계만 설명하기로 한다. 첫 번째 단계는 우

선 순위(priority)를 무시하고 의미를 설명한 뒤 두 번째 단계에서 우선 순위를 고려하는 것이다. 다음은 시간-소모 액션과 순간적인 이벤트에 대한 법칙이다.

$$\text{ActT}\ \frac{}{E \triangleright A:P \xrightarrow{A} E \triangleright P}\ (\text{단}, A \subseteq E)$$

$$\text{ActI}\ \frac{}{E \triangleright (a,n).P \xrightarrow{(a,n)} E \triangleright P}$$

예를 들면, 프로세스 $E \triangleright \{(r_1,p_1,n_1),(r_2,p_2,n_2)\}:P$는 하나의 시간 단위 동안 n_1개의 부분자원을 갖는 공유자원 r_1과 n_2개의 부분자원을 공유자원 r_2를 동시에 사용하고, P를 수행한다. 여기서, 조건인 $A \subseteq E$의 의미는 액션 A에서 사용되어지는 자원은 환경 E에 속해있는 공유자원이라는 것을 의미한다. 마찬가지로 프로세스 $(a,n)P$는 이벤트 (a,n)를 수행하고 P로 진행한다.

다음은 두 프로세스 중에서 선택을 나타내는 법칙이며 이것은 시간 소모 액션과 순간적 이벤트에 동일하게 적용된다.

$$\text{ChoiceL}\ \frac{E \triangleright P \xrightarrow{a} E \triangleright P'}{E \triangleright P+Q \xrightarrow{a} E \triangleright P'}$$

$$\text{ChoiceR}\ \frac{E \triangleright Q \xrightarrow{a} E \triangleright Q'}{E \triangleright P+Q \xrightarrow{a} E \triangleright Q'}$$

프로세스의 병렬성은 다음과 같은 법칙으로 설명이 된다. 우선 첫 번째 법칙은 두 개의 시간-소모 전이에 관한 것이다.

$$\text{ParT}\ \frac{E \triangleright P \xrightarrow{A_1} E \triangleright P', E \triangleright Q \xrightarrow{A_2} E \triangleright Q'}{E \triangleright (P \| Q) \xrightarrow{A_1 \cup A_2} E \triangleright (P' \| Q')}\ (A_1 \cup A_2 \subseteq E)$$

여기서 시간-소모 전이는 완전히 동기적(synchronous)이며, 조건$(A_1 \cup A_2 \subseteq E)$은 정의된 자원을 각 프로세스가 사용할 수 있음을 나타낸다. 여기서도 액션 A에 사용되는 자원은 환경 E에서 정의된 공유자원이다.

그리고 다음의 세 가지 법칙은 이벤트 전이에 대한 것으로 시간-소모 전이와 달리 이벤트는 비동기적(asynchronous)으로 발생할 수 있다.

$$\text{ParIL}\ \frac{E \triangleright P \xrightarrow{(a,n)} E \triangleright P'}{E \triangleright (P \| Q) \xrightarrow{(a,n)} E \triangleright (P' \| Q)}$$

$$\text{ParIR}\ \frac{E \triangleright Q \xrightarrow{(a,n)} E \triangleright Q'}{E \triangleright (P \| Q) \xrightarrow{(a,n)} E \triangleright (P \| Q')}$$

$$\text{ParCom}\ \frac{E \triangleright P \xrightarrow{(a,n)} E \triangleright P', E \triangleright Q \xrightarrow{(\bar{a},m)} E \triangleright Q'}{E \triangleright (P \| Q) \xrightarrow{(\tau,n+m)} E \triangleright (P' \| Q')}$$

처음의 두 법칙은 이벤트가 임의로 인터리빙(interleaving)되어 수행됨을 보여주고, 마지막 법칙은 두 개의 동기적 프로세스에 관한 것이다. 즉, P는 레이블 a로 이벤트를 수행하고, Q는 역 레이블 $\bar{a}$로 이벤트를 수행한다. 이와 같이 두 개의 이벤트가 동기되어 내부 τ액션으로 변환되며, 결과적으로 우선 순위는 두 개의 우선 순위를 더한 것만큼 커진다. 여기서 레이블 $\bar{a}$를 수행할 수 없으면 동기화가 이루어지지 않는다는 것이다.

제한 연산자는 시스템의 행동으로부터 제외되는 순간적 이벤트의 부분 집합을 정의한다. 이것은 $F(\tau \notin F)$에 속한 레이블들의 집합을 정해 놓고 이 레이블들에 포함되지 않는 이벤트들의 행동만을 이끌어냄으로써 이루어진다. 시간-소모 액션은 영향을 받지 않고 그대로 남는다.

$$\text{ResT}\ \frac{E \triangleright P \xrightarrow{A} E \triangleright P'}{E \triangleright P \setminus F \xrightarrow{A} E \triangleright P' \setminus F}$$

$$\text{ResI}\ \frac{E \triangleright P \xrightarrow{(a,n)} E \triangleright P'}{E \triangleright P \setminus F \xrightarrow{(a,n)} E \triangleright P' \setminus F}\ (a, \bar{a} \notin F)$$

제한 연산자가 프로세스들에 전용인 포트를 부여하는 반면, 폐쇄 연산자는 전용의 자원을 부여한다. 어떤 프로세스 P가 $[P]_I$와 같은 폐쇄된 구문(closed context) 안에 있을 때, 자원 I에 대한 더 이상의 공유는 없다.

$$\text{CloseT}\ \frac{E \triangleright P \xrightarrow{A_1} E \triangleright P}{E \triangleright [P]_I \xrightarrow{A_1 \cup A_2} E \triangleright [P']_I}$$
$$(A_2 = \{(r,0) \mid r \in I - \rho(A_1)\}, I \subseteq E)$$

$$\text{CloseI}\ \frac{E \triangleright P \xrightarrow{(a,n)} E \triangleright P}{E \triangleright [P]_I \xrightarrow{(a,n)} E \triangleright [P']_I}$$

연산자 $recX.P$는 반복(recursion)을 나타내며, 무한히 반복적인 동작을 표현한다.

$$\text{Rec}\ \frac{E \triangleright P[recX.P] \xrightarrow{a} E \triangleright P'}{E \triangleright recX.P \xrightarrow{a} P'}$$

여기서, $P[recX.P]$는 P내의 X의 발생에 대해 $recX.P$로 대체하는 표준적 표기법을 나타낸다.

두 ACSMR 프로세스의 등가(equivalence)는 우선 순위의 개념이 첨가된 강한 바이시뮬레이션(strong bisimulation)이라는 개념에 의해 결정된다. 이 개념은 만약 프로세스 $E \triangleright P$(또는 Q)가 액션 a를 하여 한 스텝 진행하면 $E \triangleright Q$(또는 P) 또한 액션 a를 하며 한 스텝 진행이 되어야 하며, 다음의 프로세스 역시 바이시뮬

러(bisimilar)하다는 의미이다.

4. 적용 예 1 : Early-Deadline-First

4.1 Early-Deadline-First 기법 명세

다중 프로세서에서 EDF로 스케줄링되는 태스크 시스템을 ACSMR로 정확한 명세를 한 후에 EDFSys와 스케줄링되는 태스크 시스템이 약한(weakly) 바이시뮬러함을 보임으로서 스케줄가능성(schedulability)을 분석할 수 있다.

여기서는 두 개의 CPU하에서 EDF 알고리즘에 의해서 태스크에 대해서 살펴보았다. 우선 모든 태스크들은 두 개의 CPU에서만 실행이 되므로, CPU들을 환경(E)로 정의하여, 각 태스크들의 실행을 제한하여 시스템을 아래와 같이 명세 하였고, 해당하는 ACSMR 표현은 [그림 1]과 같다.

EDF 알고리즘을 설명하기 위해서 $T_1, ..., T_n$의 태스크(task) 집합이 있고, 각각 데드라인인 $d_1, ..., d_n$과 수행주기 $p_1, ..., p_n$, 그리고 수행시간 $c_1, ..., c_n$을 갖고 모두 시간 0에 수행을 시작한다고 가정한다.

EDF에 의해 스케줄되는 태스크 시스템 전체를 묘사하기 위해 Job 프로세스와 Activator 프로세스를 결합한다. Job과 Activator가 병렬로 수행되는 것이 하나의 태스크 프로세스이며, 여러 개의 태스크 시스템이 병렬로 수행되는 것이 전체 시스템인 EDFSys이다. 그리고 d_{max}를 $(1+max\{d_1, ..., d_n\})$으로 정의하면 각 태스크 T_i에 대한 우선순위를 다음과 같이 표현할 수 있다.

$$\pi_i = d_{max} - (d_i - t)$$

여기서 $i = 1, ..., n$이고 t는 경과된 시간에 대한 변수이다. π_i는 t가 증가할수록 커지며 데드라인이 가까이 있는 태스크의 우선순위가 커지고 가장 큰 우선순위를 갖는 태스크를 선택하여 실행시킬 수 있다.

[그림 1]은 EDF 알고리즘에 의한 멀티프로세서의 태스크 스케줄링을 ACSMR로 표현한 것이다. EDFSys는 T_1부터 T_n까지 n개의 태스크로 구성되어 있으며 이 태스크들이 동시에 수행되는 것을 나타낸다. 각 태스크는 Job과 Activator가 *start*와 *end*를 통하여 동기화 통신을 한다. 즉 각 Job들은 Activator의 *start* 시간에 시작하며, *end* 시간에 끝난다. Exec은 s가 수행시간 c보다 작을 때 수행이 되고 c와 같을 때 Wait상태로 바뀌게 된다. n개의 태스크들 중에서 데드라인(deadline)에 가까운 것일수록 보다 높은 우선 순위를 가지고 먼저 실행되게 된다.

$\mathbf{c}_i$: computation time of the task $\mathbf{T}_i$

Constants : $\mathbf{p}_i$: period of the task $\mathbf{T}_i$

$\mathbf{d}_i$: relative deadline of the task $\mathbf{T}_i$ $(i = 1, , n)$

$\mathbf{d}_{max}$: $(1+\max\{\mathbf{d}_1, , \mathbf{d}_n\})$

$$E \triangleright EDFSys \stackrel{def}{=} [\mathrm{T}_1 \| \mathrm{T}_2 \| \cdots \| \mathrm{T}_n]_{:cpu:} \quad E = \{(2, cpu)\}$$

$$\mathrm{T}_i \stackrel{def}{=} (\mathrm{Job}_i \| \mathrm{Activator}_i) \setminus \{start, end\}$$

$$\mathrm{Job}_i \stackrel{def}{=} \varnothing : \mathrm{Job}_i + (start?,1).\mathrm{Exec}_i(0,0)$$

$$\mathrm{Exec}_i(s,t) \stackrel{def}{=} (s < c_i) \rightarrow \{(cpu, d_{\max} - (d_i - t))\} : \mathrm{Exec}_i(s+1, t+1) + \varnothing : \mathrm{Exec}_i(s, t+1) + (s = c_i) \rightarrow \mathrm{Wait}_i$$

$$\mathrm{Wait}_i \stackrel{def}{=} \varnothing : \mathrm{Wait}_i + (end?,1).\mathrm{Job}_i$$

$$\mathrm{Activator}_i \stackrel{def}{=} (start!,1).\varnothing^{d_i} : (end!,2).\varnothing^{p_i - d_i} : \mathrm{Activator}_i$$

그림 1 Earliest-Deadline-First(EDF)에 대한 ACSMR 명세

4.2 Early-Deadline-First 기법의 검증

명세가 작성된 실시간 시스템이 정확한지 검증하기 위해서 스케줄가능성 분석을 수행한다. 스케줄가능성 분석은 특정한 스케줄링 기법에 의해 스케줄되는 실시간 시스템이 데드라인 내에 일을 끝낼 수 있는지를 검사하는 것이다, 어떤 시스템이 주어진 데드라인 내에 일을 끝내면 그 시스템은 스케줄가능 하다고 한다. 그러므로, ACSMR로 명세된 실시간 시스템이 스케줄 가능한 것을 보임으로써 그 시스템이 정확한지를 검증할 수 있다, 주기적으로 반복되는 시스템에서 어떠한 데드라인도 미스되지 않는다면(즉, 데드라인을 만족하며 수행된다면) 그 시스템은 같은 동작을 무한히 반복하게 된다. 만약 모든 자원을 숨기고 내부 동작을 무시하면 무한히 반복되는 어떠한 ACSMR 프로세스도 무한히 반복되는 아이들링 액션, 즉, 프로세스 $\varnothing^{\infty}$와 등가이다. 결국 ACSMR로 묘사된 어떤 실시간 시스템이 스케줄가능한지는 $\varnothing^{\infty}$와 등가인지, 즉 약한 바이시밀러한지를 검사함으로써 확인할 수 있다.

5. 적용 예 2 : 수퍼스칼라 프로세서 ToyP

5.1 ToyP 프로세서

다중 포트 레지스터는 수퍼스칼라 프로세서에서 명령어의 병렬 실행을 효과적으로 향상시키기 위해 개발된

마이크로 프로세서 구조이다. 수퍼스칼라 프로세서는 동시에 여러 명령어를 실행 유니트에 배출(issue)할 수 있는 구조를 가지고 있으나 모든 프로그램에서 최대한의 명령어 병렬성을 얻을 수는 없다. 그 이유는 명령어들 사이의 데이타 의존성 때문에 같은 레지스터를 읽고 쓰는 명령어들 사이에는 그 실행 순서가 유지되어야 하며, 분기 명령어 같은 명령어 실행 순서가 바뀌는 경우가 있기 때문이다. 다중 포트 레지스터를 채용한 경우 여러 명령어가 한 레지스터에 대해 읽기 접근을 수행할 수 있으므로 조금이라도 병렬성을 높일 수 있다[4, 5, 6]. 그러나 다중 포트라고 해도 쓰기 접근에 대해서는 공유되어서는 안되기 때문에 그 경우는 모든 포트가 한 명령어에 독점된다. 여기서는 ToyP 라는 32비트 가상 수퍼스칼라 프로세서의 명령어를 사용하였다.

예제를 위해 [13] 에서와 같이 ToyP라는 가상적인 32비트 수퍼스칼라 프로세서의 명령어를 사용할 것이다. ToyP 프로세서는 많은 상업적인 프로세서들의 특징을 갖추고 있으며, 앞서 말한 Harcourt 등에 의해 개발되었다. 그들은 일찌기 명령어 명세를 위해 SCCS라는 프로세스 대수를 이용했었다[10]. SCCS에는 자원이란 개념이 없었으므로 각 자원을 바이너리 세마포어로 나타내어, 제시된 명세는 매우 복잡하고 거추장스러웠다. 더군다나 SCCS는 우선순위 개념 또한 없어 CCS를 위해 개발된 우선순위 연산자를 끌어 들여야 했다. 이러한 파이프라인 실행의 모델링에 대한 기존 연구로는 최초로 파이프라인 실행 행태를 반영하여 프로그램의 최악 실행시간을 분석하고자 한 연구는 Zhang 등의 연구[7]가 있다. 이 연구에서는 Intel 80C188 프로세서의 두 단계 파이프라인 실행을 모델링하여 명령어의 중첩된 실행 행태를 최악 실행시간 분석에 반영하였지만, 모델로 삼은 Intel 80C188 프로세서는 두 단계만으로 이루어진 지극히 간단한 파이프라인 실행 구조를 갖추고 있어서 이 모델을 일반적인 파이프라인 실행 구조에 적용하기는 어렵다. 서울대학교 실시간 시스템 연구 그룹의 Lim 등이 제안한 확장된 타이밍 스키마(timing schema)[8]에서는 자원예약표를 이용하여 파이프라인 실행을 모델링하였다. 자원예약표는 명령어가 파이프라인 실행되는 모습을 파악하기 위해 제안된 기법으로 실행시간의 흐름에 따른 각 실행자원의 이용 상태를 표의 형태로 나타낸다. 플로리다 대학의 Healy 등은 Lim 등의 연구에서 사용한 자원예약표와 같은 형태의 파이프라인 실행도(pipeline diagram)[9]를 사용하여 파이프라인 실행을 모델링하였다. 프린스턴 대학의 Li 등은 프로그램을 정수 선형 계획법을 사용해 표현하기 위해 각 기본 블록의 최악 실행시간을 상수로 결정하여 사용하였다. Li 등의 연구와 Zhang 등의 연구에서는 기본 블록 사이의 파이프라인 실행으로 인한 중첩 실행을 반영하지 않았기 때문에 분석 대상 프로그램의 특성에 따라서는 분석 결과와 실제 실행시간 사이에 큰 차이가 발생할 수 있지만[18], 여기서 제안된 방법은 일반적인 명령어 실행 모델링을 정형 검증하여 정확한 분석을 할 수 있다. [표 1]은 [11]에서 모델링한 ToyP 명령어에다 [15]에서 추가한 간단한 분기 명령어를 추가해서 나타내었다

표 1 사용된 ToyP 명령어

Add Ri, Rj, Rk	Ri ← Rj + Rk
Mov Ri, Rj	Ri ← Rj
Load Ri, Rj, #c	Ri ← Mem[Rj + c]
Store Ri, Rj, #c	Mem[Rj + c] ← Ri
Jmp #c	PC ← c

보통 Add와 Mov명령어는 한 명령어 사이클에 연산이 실행된다. 반면 메모리와 관련된 명령어인 Load, Store는 실행되는데 두 사이클이 필요하다. Jmp는 다른 어떤 명령어와도 동시에 실행되지 못하며 프로그램 카운터(PC)를 분기 목적지를 가르치도록 한다.

ToyP의 전체 시스템은 우선 순위가 전부 같은 레지스터들의 유한 집합 R로 구성되어 있다고 생각한다. 하나의 액션은 R의 부분집합으로 정의되며 한 사이클 타임이 걸린다. 그러나 여기서 자원은 여러 개의 부분 자원으로 나누어져 있는 것으로 생각되므로 하나의 자원을 접근하는데 그 부분 자원을 얼마나 사용하는 것인지 나타낼 필요가 있다. 이것은 우선 순위 다음에 역시 순서쌍(tuple)으로 나타낸다. 예를 들어 원소가 하나인 부분집합, $\{(r, i, j)\}$ 은 어떤 레지스터 $r \in R$을 우선 순위 i로 j개만큼 사용하고 한 사이클을 소모하는 액션을 나타낸다. 명료성을 위해 이 다음부터는 액션에서 우선 순위가 모두 같다고 생각하고 빼도록 하겠다. 그리고 액션 $\varnothing$(또는 {})은 한 시간 단위 동안 아무 것도 하지 않는 것을 나타내며, A, B, C는 액션을 나타내는 문자이다.

정의 5.1 명령어 실행의 종료를 나타내는 프로세스 Done을 다음과 같이 정의한다.

$$\text{Done} \stackrel{def}{=} recX \,.\varnothing : X$$

프로세스 Done은 병렬 연산자 에 대해 다음과 같은

특성이 있다. 모든 프로세스 P에 대해 P || Done = P 이다.

다음의 이진 연산자 next는 이어지는 사이클에 명령어를 배출(issue)하는 것을 모델링한다. 자원 J는 분기 명령어를 검색하기 위한 것으로 분기 명령어는 똑같이 자원 J를 사용함으로써 *next* 연산자와 충돌하여 일반적으로 순서적인 명령어 배출이 이루어지지 않도록 한다. J는 다중 자원이 아니므로 [그림 2]에서 환경 E에 최대 부분 자원이 1로 한정되어 있다.

$$\mathrm{Insts(PC)} \stackrel{def}{=} \varnothing : \mathrm{Insts(PC)} + \mathrm{Super-Insts(PC)}$$

$$\mathrm{Super-Insts(PC)} \stackrel{def}{=} (\mathrm{Mem(PC)} \,\|\, \mathrm{Mem(PC+4)} \,\|\, \mathrm{Mem(PC+8)})\ \mathit{next}\ \mathrm{Insts(PC+12)} + (\mathrm{Mem(PC)} \,\|\, \mathrm{Mem(PC+4)})\ \mathit{next}\ \mathrm{Insts(PC+8)} + (\mathrm{Mem(PC)}\ \mathit{next}\ \mathrm{Insts(PC+4)} + \mathrm{Mem(PC)}$$

$$\mathrm{Program} \stackrel{def}{=} [\mathrm{Super-Insts(PC)}]_R$$

$$E \rhd \mathrm{Program} \qquad E = \{(6, \mathrm{R}i) \mid 1 \le i \le 6\} \cup \{(1, \mathrm{J})\}$$

그림 2 ToyP 프로세서의 실행 모델링

정의 5.2 임의의 P, Q에 대해, P *next* Q = P || {(J,1)}:Q.

5.2 ACSMR을 이용한 ToyP 명령어 모델링

ToyP 프로세서의 정수 레지스터에는 두 가지 종류의 동작(operation)이 있다. 읽기와 쓰기가 그것인데, 값을 읽을 때는 다중 포트를 통해 레지스터를 공유할 수 있으나 쓸 때는 한번에 한 명령어 프로세스만이 레지스터를 사용할 수 있다.

ACSMR 자원은 동시에 복수개의 부분 자원으로서 재사용할 수 있으므로, 각 레지스터는 하나의 자원으로 표현되며, 환경 E에서 포트의 수 만큼 다중 자원의 최대 수를 제한 받는다. 포트의 수는 얼마나 많은 명령어가 병렬로 읽혀 실행되어 질 수 있느냐에 달려 있다. ToyP는 데이타 해저드가 없다면 한 사이클에 세 개의 정수 명령어를 읽어 들여 실행할 수 있는 것으로 가정한다면, 각 정수 명령어는 한 레지스터를 두번 읽을 수 있으므로 (**Add** R2, R1, R1에서처럼) 각 레지스터는 한 사이클에 여섯번의 읽기 요구를 받을 수 있다. 한 명령어 프로세스가 값을 읽으려고 할 때는 여섯 개의 포트 중 어느 하나만 얻으면 되지만 값을 쓸 때는 포트 여섯 개를 전부 얻어야 한다. 그러므로 한 프로세스가 쓰기를 위해 레지스터를 잡고 있으면 다른 프로세스는 레지스터를 공유할 수 없다. [표 2]에서 (Ri,1)은 명령어가 실행되는 그 때에 유효한 레지스터 i의 포트 하나를 사용하여 읽기 접근을 함을 나타내는 것이다. 그러므로 (Ri,6)은 레지스터 i의 모든 자원을 독점하는, 즉 쓰기 접근을 나타낸다.

[표 2]에 나와 있는 대로 ACSMR을 이용하여 ToyP 명령어를 모델링한다. [표 2]에 의해 얻어진 ToyP 프로그램에 대한 ACSMR 프로세스의 집합을 프로그램 명세라고 한다. 다음 예제는 ToyP 프로그램을 프로그램 명세로 번역하는 방법을 보여 준다.

표 2 ACSMR로 표현된 명령어

$$\mathrm{Add\ R}i, \mathrm{R}j, \mathrm{R}k \stackrel{def}{=} \{(\mathrm{R}i,6),(\mathrm{R}j,1),(\mathrm{R}k,1)\} : \mathrm{Done}$$

$$\mathrm{Mov\ R}i, \mathrm{R}j \stackrel{def}{=} \{(\mathrm{R}i,6),(\mathrm{R}j,1)\} : \mathrm{Done}$$

$$\mathrm{Load\ R}i, \mathrm{R}j, \#\mathrm{c} \stackrel{def}{=} \{(\mathrm{R}i,6),(\mathrm{R}j,1)\} : \{(\mathrm{R}i,6)\} : \mathrm{Done}$$

$$\mathrm{Store\ R}i, \mathrm{R}j, \#\mathrm{c} \stackrel{def}{=} \{(\mathrm{R}i,1),(\mathrm{R}j,1)\} : \{(\mathrm{R}i,1)\} : \mathrm{Done}$$

$$\mathrm{Jmp}\ \#\mathrm{c} \stackrel{def}{=} \{(\mathrm{J},1)\} : \mathit{Insts}(\mathrm{c})$$

예제 5.1 메모리 위치 PC에 다음 프로그램이 있을 때,

PC: **Add** R1, R1, R1
PC+4: **Jmp** #(PC+12)
PC+8: **Load** R2, R3, #8
PC+12: **Add** R1, R3, R3

이는 다음의 ACSMR 프로세스로 나타내어 질 수 있다.

$$\mathrm{Mem(PC)} \stackrel{def}{=} \{(R1,6)\} : \mathrm{Done}$$

$$\mathrm{Mem}(PC+4) \stackrel{def}{=} \{(\mathrm{J},1)\} : \mathrm{Insts(PC+12)}$$

$$\mathrm{Mem}(PC+8) \stackrel{def}{=} \{(R2,6),(R3,1)\} : \{(R2,6)\} : \mathrm{Done}$$

$$\mathrm{Mem(PC+12)} \stackrel{def}{=} \{(R1,6),(R3,1)\} : \mathrm{Done}$$

$$\mathrm{Mem(PC+16)} \stackrel{def}{=} \mathrm{NIL}$$

5.3 ACSMR을 이용한 ToyP 실행 모델링

데이타 해저드를 가진 모든 명령어는 ACSMR 식에서 같은 자원을 사용하는 프로세스로 묘사되며 같이 실행될 경우 충돌하여 실행될 수 없는 프로세스가 된다. 예를 들어 같은 레지스터 Ri에 쓰기를 하는 명령어와 읽기를 하는 명령어가 있다면 쓰기 명령어의 경우는 레지스터 포트를 전부 독점하므로 {(Ri,6)}의 액션을 하는

ACSMR 프로세스가 될 것이고 읽기 명령어는 레지스터 포트를 하나만 사용하므로 {(R*i*,1)}의 액션을 하는 ACSMR 프로세스로 표현될 것이다([표 2]). 이 두 프로세스가 동시에 수행되면 소비하는 자원의 이름이 같으므로 부분 자원의 개수가 합쳐져 {(R*i*,7)}의 액션을 수행하게 되고 이는 환경에서 지정된 범위(≤ 6)를 벗어나는([그림 2]) 것이어서 NIL이 된다. 이런 식으로 데이타 해저드를 가진 모든 명령어 프로세스는 NIL이 되어 명령어 모델링에서는 전부 제거되게 된다. ACSMR의 이러한 성질은 타이밍 분석을 용이하게 하고 최종 명세에서 실제로 사용되어지는 레지스터 자원의 개수를 정확히 산정할 수 있게 해 준다.

ToyP 프로그램의 실행은 각 명령어 주기에서 가장 많은 자원이 소비될 수 있는 쪽으로 진행되게 된다[13]. 이 과정을 다음 예제 5.2에서 자세히 살펴보도록 하겠다. 예제 5.2는 동시에 실행되는 프로세스끼리 어떻게 자원의 공유가 일어나는지 잘 보여 주고 있다..

예제 5.2 이 예제는 예제 5.1에 나와 있는 ToyP 프로그램 명세를 이용하여 확장하는 과정을 보여 주고 있다.

Program

$$= \left[\begin{array}{l} \mathrm{Mem(PC) \| Mem(PC+4) \| Mem(PC+8)}\ \mathit{next}\ \mathrm{Insts(PC+12)+} \\ \cdots + \mathrm{Mem(PC+4)}\ \mathit{next}\ \mathrm{Insts(PC+8)+Mem(PC+4)} \end{array}\right]_R$$

$$= \left[\begin{array}{l} \{(\mathrm{R1,6}),(\mathrm{R2,6}),(\mathrm{R3,1}),(\mathrm{J,2})\}:\{(\mathrm{R2,6})\}:\mathrm{Done} \| \mathrm{Insts(PC+12)} \| \cdots + \\ \cdots \{(\mathrm{R1,6}),(\mathrm{J,1})\}:\mathrm{Insts(PC+8)} + \{(\mathrm{R1,6})\}:\mathrm{Done} \end{array}\right]_R$$

$$= [\mathrm{NIL} + \mathrm{NIL} + \{(\mathrm{R1,6}),(\mathrm{J,1})\}:\mathrm{Insts(PC+8)} + \{(\mathrm{R1,6})\}:\mathrm{Done}]_R$$

$$= [\{(\mathrm{R1,6}),(\mathrm{J,1})\}:\mathit{Insts}(\mathrm{PC+8})]_R$$

첫 번째 주기에서 충돌하는 프로세스를 모두 제거하면 결국 처음 명령어만이 실행될 수 있음을 알 수 있다. 다음 주기에서는 분기 명령어가 있으므로 바로 네 번째 명령어로 건너가서 실행되도록 한다. 최종적인 확장식은 다음과 같이 된다.

$$\mathrm{Program} = [\{(\mathrm{R1,6})\}:\{(\mathrm{J,1})\}:\{(\mathrm{R1,6}),(\mathrm{R3,1})\}:\mathrm{Done}]_R$$

6. 결 론

실시간 시스템에서는 정확성이 매우 중요하므로 시스템을 설계할 때 그 정확성을 보장해 주는 방법이 필요하다. 이 논문에서 제시한 것처럼 동시성, 우선 순위, 시간등과 같이 실시간 시스템을 표현하는데 필요한 개념을 애매모호함 없이 제공하는 정형기법을 이용함으로써 시스템의 행동을 정확히 묘사할 수 있다.

위에서 언급한 것과 같이 다중자원을 사용하는 ACSMR을 이용하여서 멀티프로세서 하에서 Earliest-Deadline-First를 스케줄링 기법으로 사용하는 실시간 시스템을 정확히 묘사하여, 명세, 검증함으로써 같은 작업에 대해서 단일프로세서를 사용하는 경우와 멀티프로세서를 사용하는 경우에 대한 시스템 행동에 대한 정확한 차이를 조사할 수 있었고, ToyP의 경우에는 어떤 레지스터에 포트가 모두 몇 개가 쓰였으며 어떤 명령어가 실행되었는지, 그리고 구현된 프로세서 모델이 제대로 작동하는지 알 수 있다.

기존에 ACSR을 사용한 경우와 비교하면 ACSR에서는 부분자원을 사용하는 경우를 모두 하나의 자원을 사용하는 것으로 명시하여만 하므로 하나의 자원에 대해 여러 가지의 경우로 반복이 일어나고 나중에 이 부분을 모두 합쳐주어야 하지만, ACSMR에서는 하나의 자원에 대해 부분 자원을 사용하는 것으로 명시함으로 이런 번거로움을 피할 수 있는 장점이 있다.

참 고 문 헌

[1] R.Milner, *Communication and Concurrency*. Prentice Hall, 1989.

[2] Insup Lee, Patrice Bremond-Gregorie, and Richard Gerber, "A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems," *Proceedings of the IEEE*, pp. 158-171, Jan. 1994.

[3] T. Cook, P. franzon, E. Harcourt, and T. Miller. "System-Level Specification of Instruction Sets," *Proc. of the Internatinal Conference on Computer Design*, 1993.

[4] D. Patterson and J. Hennesy. *Computer Architecture: A Quantitative Approach*, Morgan Kauffman, 1990.

[5] M. Johnson. *Superscalar Microprocessor Design*, Prentice-Hall, 1991.

[6] R. Rau and J. Fisher. "Instruction-Level Prallel Prodessing: History, Overview, and Perspective," *Journal of Supercomputing*, July, 1993.

[7] N. Zhang, A. Burns, and M. Nicholson, "Pipelined Processors and Worst-Case Execution Times," *Real-Time Systems*, Vol. 5, No. 4, pp. 319-343, Oct. 1993.

[8] S. -S. Lim, Y. H. Bae, G. T. Jang, *etc.*, "An Accurate Worst Case Timing Analysis for RISC Processors," *IEEE Transactions on Software Engineering*, Vol. 21, No. 7, pp. 593-604, July 1995.

[9] C. A. Healy, D. B. Whalley, and M. G. Harmon, "Integrating the Timing Analysis of Pipelining and Instruction Caching," *Proceedings of the 16th*

Real-Time Systems Symposium, pp. 288-297, Dec. 1995.
[10] E. Harcourt, J. Mauney, and T. Cook. "Specification of Instruction-Level Parallelism," In *Proc. of the North American Process Algebra Workshop*, 1993.
[11] Jin-Young Choi, Insup Lee, and Inhye Kang. "Timing Analysis of Superscalar Processor Programs Using ACSR," *IEEE Real-Time Systems Newsletter*, Vol. 10, No. 1/2, 1994.
[12] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, Vol. 39, No. 9, Sept. 1990.
[13] Hanene Ben-Abdallah, Jin-Young Choi, Duncan Clarke, Young Si Kim, Insup Lee, and Hong-Liang Xie, "A Process Algebraic Approach to the Schedulability Analysis of Real-Time Systems," *Real-Time Systems Journal*, to appear.
[14] 임성묵, 최진영, "ACSR-VP를 이용한 Priority Ceiling Protocol의 명세와 검증", 정보 과학회 추계 학술 대회, pp. 619-622, 1997.
[15] 이기혼, 최진영, "분기 명령어를 포함한 수퍼스칼라 프로그램의 타이밍 분석", 정보 과학회 추계 학술 대회, pp. 475-478, 1997.
[16] 이기혼, 최진영, "ACSMR을 이용한 수퍼스칼라 프로그램의 타이밍 분석", 정보 과학회 춘계 학술 대회, pp. 620-622, 1998.
[17] 유회준, 최진영, "ACSMR을 이용한 다중 프로세서에 대한 스케줄링 분석", 정보 과학회 춘계 학술 대회, pp. 593-595, 1998.
[18] 유회준, 이기혼, 최진영, "ACSMR을 이용한 수퍼스칼라 프로세서 프로그램의 비순차 정렬 명령어의 타이밍 분석", 병렬처리시스템 학술발표회, No.17, 제9권, 제2호, 1998.
[19] 이기혼, 최진영, "ACSR을 이용한 비순차 수퍼스칼라 프로세서의 시간 분석", 정보과학회 추계 학술 대회, pp. 697-699, 1998.

유 회 준
정보과학회논문지 : 시스템 및 이론
제 27 권 제 2 호 참조

이 기 혼
정보과학회논문지 : 시스템 및 이론
제 27 권 제 2 호 참조

최 진 영
정보과학회논문지 : 시스템 및 이론
제 27 권 제 2 호 참조